CS377P Homework 5
Matthew Jagen
Fall 2022

Part 1:
numPoints =1,000,000,000
Running time for sequential calculation: 20789878244 nanoseconds (20.79 seconds)

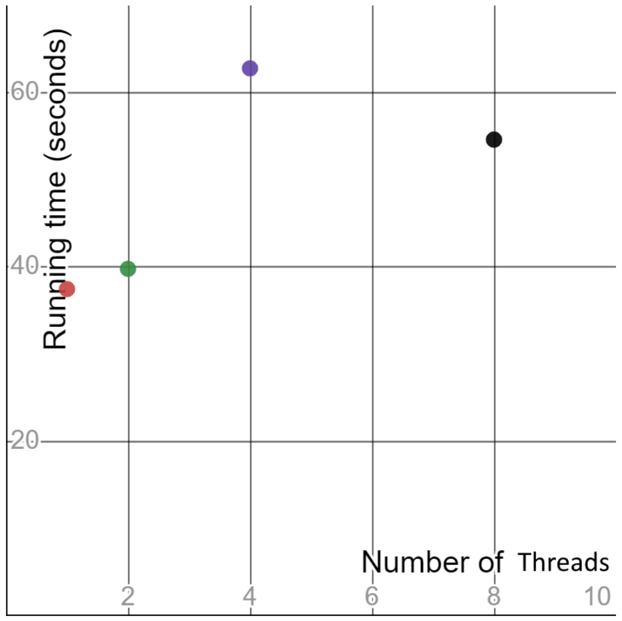
This code provides an estimate for an integral equation for pi ($\int_0^{1/2} f(x)dx = \pi$). The code estimates the value of the integral by doing a summation of many rectangular areas below the curve. I believe what it's doing is a left Riemann sum, so the estimate will always be a slight underestimate, but can be more or less accurate depending on how small/fine we make the step size.

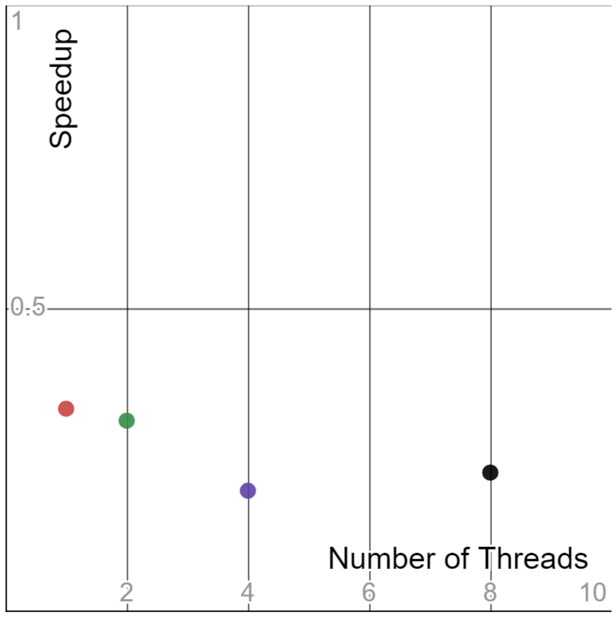
Part 2:
Running time for sequential calculation: 12558419476 nanoseconds (12.56 seconds)

The semicircle created by **y = sqrt(1 - x*x)** has a radius of 1, so it's area is (pi*1^2)/2 or just pi/2. In order for the estimate to be within 1% of the actual value, numPoints had to be 23 and the step size was 0.086957 (this gave the estimate 3.111518 for pi).

Part 3:
numPoints =1,000,000,000
1 thread:
    running time: 37480963283 nanoseconds
    speedup: 0.3351
    estimation: 3.14159265358109873745
2 threads:
    running time: 39817647800 nanoseconds
    speedup: 0.3154
    estimation: 3.01575852016901757580
4 threads:
    running time: 62795584952 nanoseconds
    speedup: 0.2000
    estimation: 2.52579650975752967312
8 threads:
    running time: 54625890409 nanoseconds
    speedup: 0.2299
    estimation: 2.19882612750193784379

As the number of threads increases, the value of pi gets less accurate, and specifically it is getting smaller and smaller. These values are not accurate because of the data race between threads writing to the global pi variable. As more threads compete to write to the variable, fewer writes actually go through and the calculated value gets much smaller in comparison.





Part 4:

numPoints =1,000,000,000

1 thread:

running time: 37716611547 nanoseconds
speedup: 0.3330
estimation: 3.14159265358109873745

2 threads:

running time: 164355074136 nanoseconds
speedup: 0.0764
estimation: 3.14159265358108319433

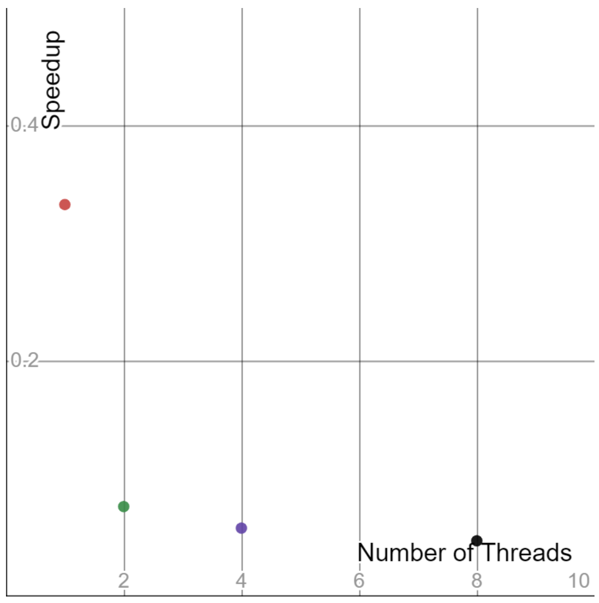
4 threads:

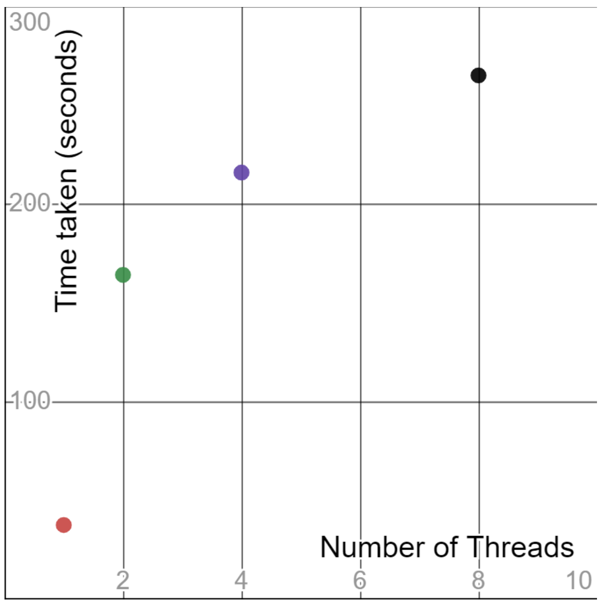
running time: 216187965597 nanoseconds
speedup: 0.0581
estimation: 3.14159265358102013366

8 threads:

running time: 265356370148 nanoseconds
speedup: 0.0473
estimation: 3.14159265358110806332

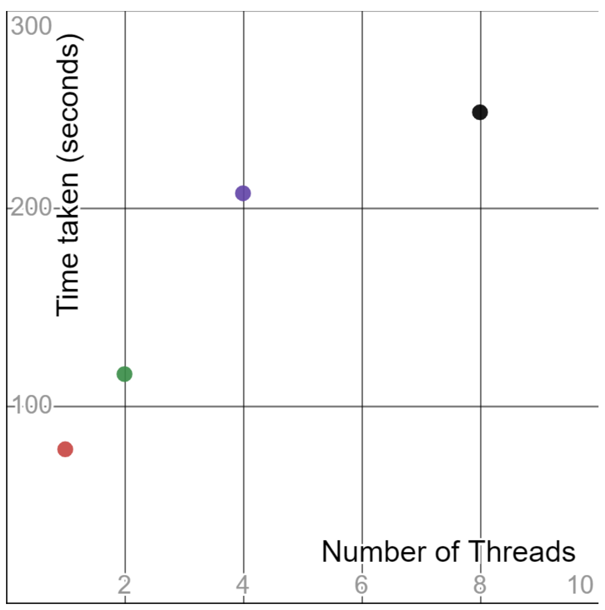
When run on 8 threads the value computed is much much closer to the actual value of pi, which is to be expected since we are avoiding the data races that were causing issues in part 3.

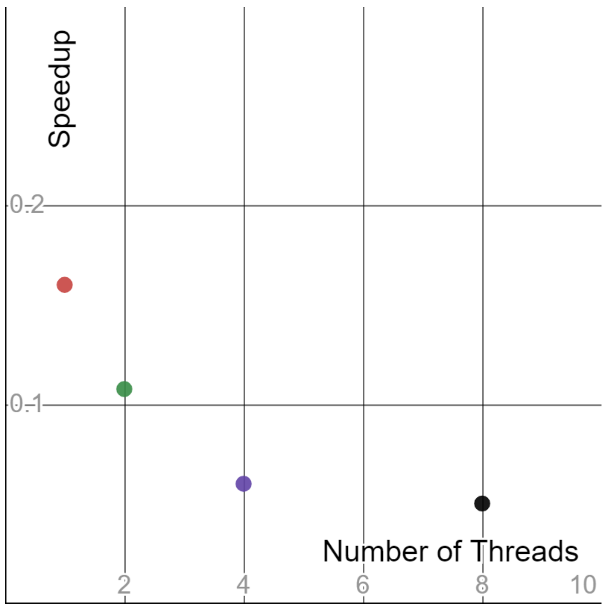




Part 5:
numPoints =1,000,000,000
1 thread:
    running time: 78288347250 nanoseconds
    speedup: 0.1604
    estimation: 3.14159265358109873745
2 threads:
    running time: 116305722804 nanoseconds
    speedup: 0.1080
    estimation: 3.14159265358109651700
4 threads:
    running time: 207772462650 nanoseconds
    speedup: 0.0604
    estimation: 3.14159265358006400959
8 threads:
    running time: 248782565098 nanoseconds
    speedup: 0.0505
    estimation: 3.14159265357984995859

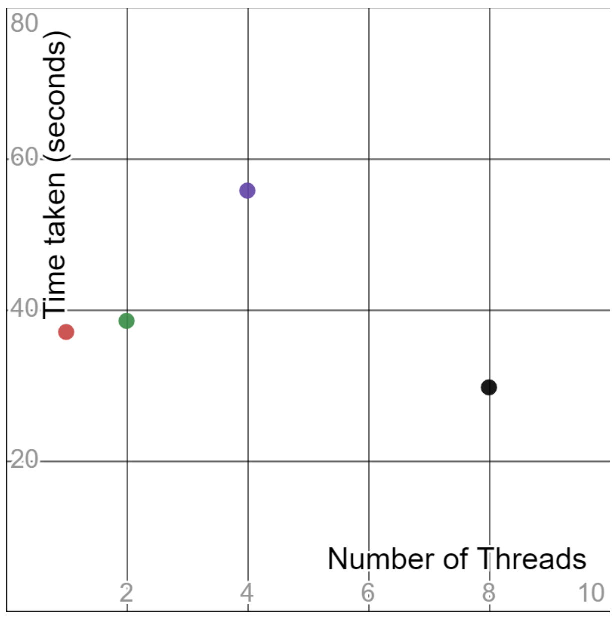
Running times seems to have sped up slightly for every one except for the one thread execution. In all, it doesn't seem much different from how we were using our mutex in part 4, since we still have to do the adds to pi serially and are bottlenecked in a similar way.

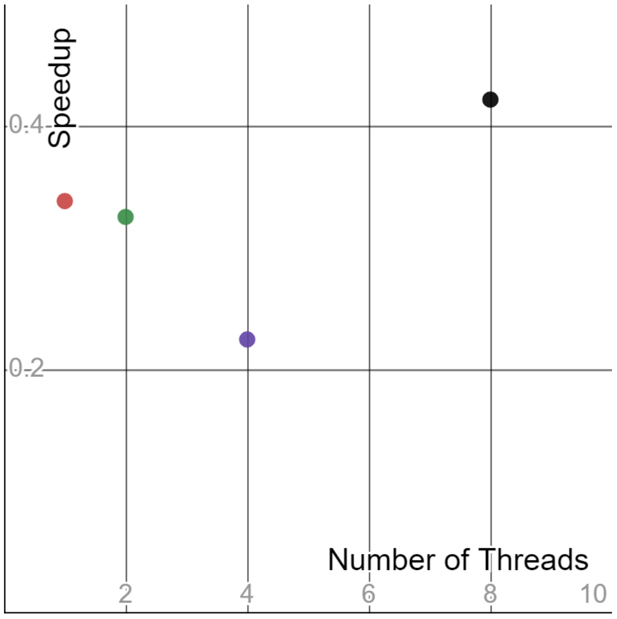




Part 6:
numPoints =1,000,000,000
1 thread:
    running time: 37094689037 nanoseconds
    speedup: 0.3386
    estimation: 3.14159265358109873745
2 threads:
    running time: 38568494390 nanoseconds
    speedup: 0.3256
    estimation: 3.14159265359128081485
4 threads:
    running time: 55803616382 nanoseconds
    speedup: 0.2250
    estimation: 3.14159265359023542885
8 threads:
    running time: 29767007249 nanoseconds
    speedup: 0.4219
    estimation: 3.14159265358970429816

The value of pi when calculated with 8 threads is still very accurate at 3.1415926535. The running time of part 6 seems to compare most similarly with the running time of part 3, but obviously with the benefit of having accurate estimates since each thread is writing to a different (but still locally close) variable.





Part 7

numPoints =1,000,000,000

1 thread:

    running time: 39107366723 nanoseconds
    speedup: 0.3211
    estimation: 3.14159265358109873745

2 threads:

    running time: 19608009772 nanoseconds
    speedup: 0.6405
    estimation: 3.14159265359128081485

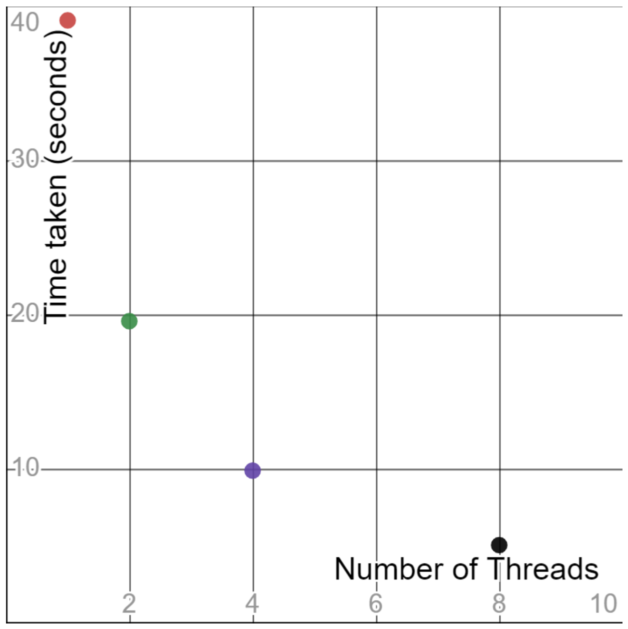
4 threads:

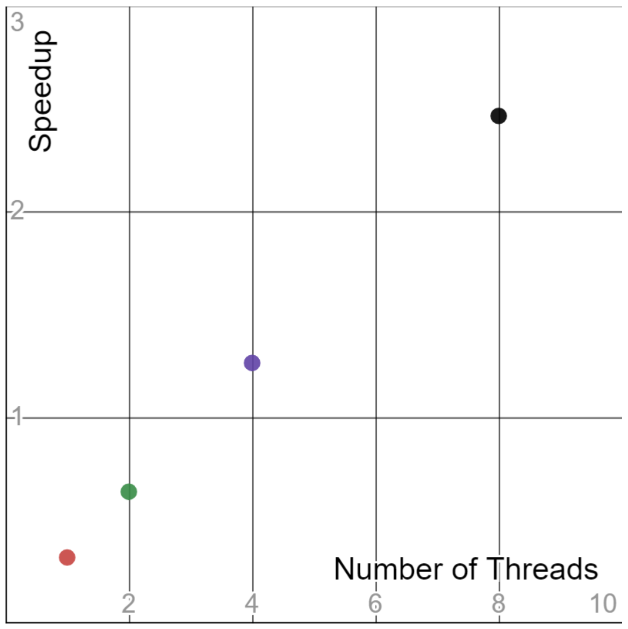
    running time: 9914050854 nanoseconds
    speedup: 1.2667
    estimation: 3.14159265359023542885

8 threads:

    running time: 5088562012 nanoseconds
    speedup: 2.4680
    estimation: 3.14159265358970429816

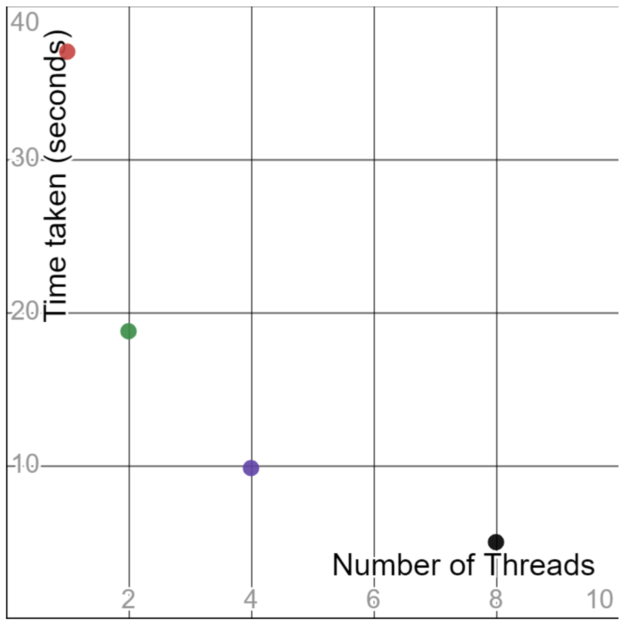
On 8 threads it is still accurate at 3.14159265358970429816.

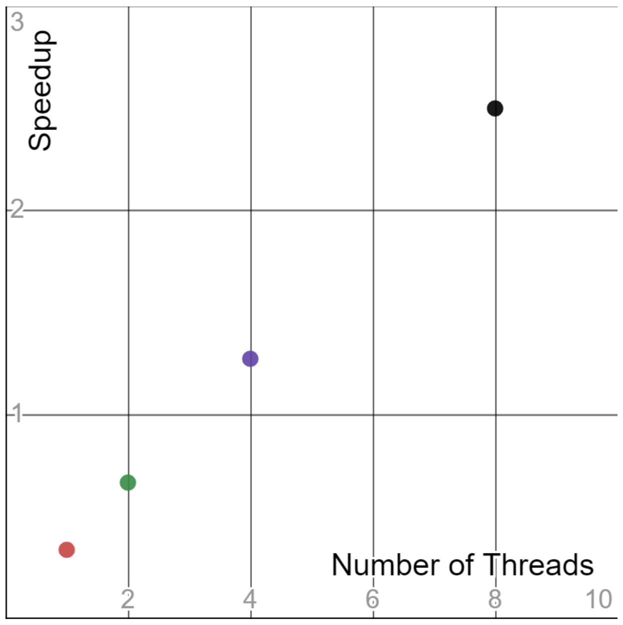




Part 8:
numPoints =1,000,000,000
1 thread:
    running time: 37058207180 nanoseconds
    speedup: 0.3389
    estimation: 3.14159265358109873745
2 threads:
    running time: 18791392097 nanoseconds
    speedup: 0.6683
    estimation: 3.14159265359128081485
4 threads:
    running time: 9854441917 nanoseconds
    speedup: 1.2744
    estimation: 3.14159265359023542885
8 threads:
    running time: 5021615240 nanoseconds
    speedup: 2.5009
    estimation: 3.14159265358970429816

On 8 threads it is still accurate at 3.14159265358970429816.





Part 9: (sorry I know it was supposed to be a ‘short’ summary)

In summary, part 3 not only took longer than parts one and two, but also got poor estimates. This is because it has data races and coherence misses. As the thread count increases, more and more threads are competing for one cache line for the global pi variable so we are heavily bottlenecked by the shared bus. Additionally, even though we go through all this trouble to keep our cache coherent, we don’t solve the issue of data races, so non-atomic load-modify-store operations can interfere with one another and cause us to have a severe underestimation in the end.

Part 4 and part 5 solve this issue by making our load-modify-store of the global variable atomic. Because we had a billion points to calculate, part 4’s mutex caused a lot of waiting and joining/leaving the queue. Part 5 didn’t use pthreads’ mutex to make it atomic, so it didn’t have quite as much overhead while serializing additions but was still very slow with parts 4 and 5 being the worst performing parts.

Part 6 gave each thread its own global variable, but since they were all on the same array, many of them fell on the same cache line and caused coherence misses with false-sharing. One interesting thing with this is that part 6 is a slight improvement over part 3 with 8 threads. I believe this is the case because instead of one cache line being ping-ponged between 8 cores, part 6 has its variables split over multiple cache lines. So, as we increase the threads, part 6 will only ever have a certain amount of cores fighting over each line.

In part 7 and part 8 we eliminated both true and false sharing, allowing us to see great improvements in performance as we increased the number of threads. Each thread had its own local variable and its own cache line, meaning we weren’t bottlenecked by the shared bus. Using pthreads’ barrier instead of join is a slight improvement in part 8, and if I had to guess, it’s because join causes the main thread to unblock and do another loop iteration after each thread terminates whereas barrier doesn’t unblock the main thread until it has the correct count of threads waiting.